



Clase 10. Python

Funciones II

***RECUERDA PONER A GRABAR LA
CLASE***





OBJETIVOS DE LA CLASE

- Reconocer los tipos de argumentos y parámetros.
- Aplicar funciones recursivas e integradas.

CRONOGRAMA DEL CURSO

Clase 9



Funciones



PAR O IMPAR



FUNCIÓN AÑO BISIESTO

Clase 10



Funciones II



RELOJ



¡FUNCIONES!

Clase 11



Excepciones

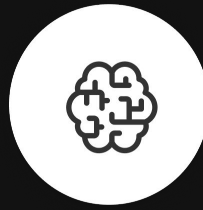


DESAFÍO DE ERRORES



DESAFÍO DE EXCEPCIONES

ARGUMENTOS Y PARÁMETROS



¡PARA PENSAR!

¿Cuál es la diferencia entre los parámetros y argumentos?

CONTESTA EN EL CHAT DE ZOOM



Argumentos y Parámetros

Como sabemos, durante la definición de la función, las variables o valores se denominan **parámetros**:

```
>>> def suma(numero1, numero2):  
    return numero1 + numero2
```

Y durante la llamada se le denominan **argumentos**, como los argumentos de los scripts.

```
>>> resultado = suma(7, 5)
```

En esta clase estaremos viendo los distintos tipos de argumentos y parámetros.



CODER HOUSE

ARGUMENTOS POR POSICIÓN



Argumentos por posición

Cuando se envían argumentos a una función, se reciben por orden en los parámetros definidos:

```
>>> def suma(numero1, numero2):  
    return numero1 + numero2  
>>> resultado = suma(7, 5)
```

El argumento **7** es la posición **0**, por consiguiente es el parámetro de la función **numero1**, seguidamente el argumento **5** es la posición **1** por consiguiente es el parámetro de la función **numero2**.



Argumentos por posición

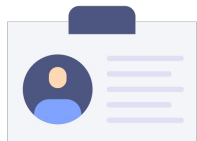
Si tomamos el siguiente ejemplo sabremos que la resta nos dará 3:

```
>>> def resta(a, b):  
    return a - b  
>>> resultado = resta(15, 12)
```

Pero, si modificamos el orden de los argumentos nos dará otro resultado:

```
>>> resultado = resta(12, 15)
```

ARGUMENTOS POR NOMBRE

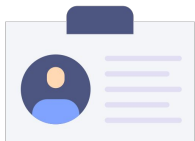


Argumentos por nombre

Como vimos, **si pasamos ordenado el argumento, se verá reflejado ordenadamente el parámetro.**

Para cambiar esto se utiliza la **asignación de argumentos por nombre**, si indicamos durante la llamada que valor tiene cada parámetro a partir de su nombre:

```
>>> def resta(a, b):  
        return a - b  
>>> resultado = resta(b=15, a=12)
```



Argumentos por nombre



Recordemos que al utilizar argumentos por nombre, no importa el orden:

```
>>> def resta(a, b, c):  
    return a - b - c  
>>> resultado = resta(a=15, b=12, c=2)
```



```
>>> resultado = resta(c=2,  
a=15, b=12)
```

LLAMADA SIN ARGUMENTOS



Llamada sin argumentos

Veamos qué pasa si llamamos una función con parámetros ya definidos:

```
>>> def resta(a, b):  
        return a - b  
>>> resultado = resta()
```

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: resta() takes exactly 2 arguments (0 given)

¿Cómo solucionamos el TypeError al momento de llamar una función sin argumento?



CODER HOUSE



Parámetros por defecto

Python, nos deja asignar unos valores por defecto a los parámetros, es decir, indicarle que tendrán un valor por defecto si no viene ningún valor.

```
>>> def resta(a=None, b=None):  
        return a - b  
>>> resultado = resta()
```




Parámetros por defecto

Como vimos con anterioridad, **no podemos restar None a None**, ya que nos devuelve un error:

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: unsupported operand type(s) for -: 'NoneType' and 'NoneType'



Para solucionar el error de restar None tendremos que hacer una validación

CODER HOUSE



Parámetros por defecto

```
>>> def resta(a=None, b=None):  
    if a == None or b == None:  
        print("Error, debes enviar dos números a la función")  
        return  
    return a - b  
>>> resultado = resta()
```

Se indica el final de la función luego de la sentencia print, usando la sentencia return aunque no devuelve nada.



BREAK

¡5/10 MINUTOS Y VOLVEMOS!

ARGUMENTOS POR VALOR Y REFERENCIA



Argumentos

Si hablamos de argumentos tenemos que tener algo en cuenta:

Cuando **enviamos información a una función** generalmente estos **datos se envían por valor**.

Eso significa que se crea una copia dentro de la función de los valores que enviamos en sus propias variables. Pero, **hay casos excepcionales**, las colecciones, listas, diccionarios, conjuntos. Estos datos **se envían por referencia**.



Referencia

¿Que significa que los conjuntos como listas, tuplas, etc, se envíen por referencia?



Significa que en lugar de una copia dentro de la función, estaremos manejando el dato original, y si lo modificamos también se verá reflejado en el exterior, es decir, en el conjunto original y no en una copia en la función. Esto debido a que hacen **referencia** a la variable externa, algo así como un acceso directo.



Dependiendo del tipo de dato:

Paso por valor: Se crea una copia local de la variable dentro de la función.

Los tipos simples se pasan por valor: Enteros, flotantes, cadenas, lógicos...

Paso por referencia: Se maneja directamente la variable, los cambios realizados dentro de la función le afectarán también fuera.

Los tipos compuestos se pasan por referencia: Listas, diccionarios, tuplas, conjuntos...



Paso por valor

Los números se pasan por valor y crean una copia dentro de la función, **no les afecta externamente** lo que hagamos con ellos en la función:

```
>>> def doblar_valor(numero):  
    numero *= 2  
  
>>> numero = 10  
>>> doblar_valor(numero)  
>>> print(numero)-----> 10
```




Paso por referencia

```
✓ [37] def doblar_valores(numeros):  
    for i,n in enumerate(numeros):  
        numeros[i] *=2
```

```
▶ #numeros ---- No se ac  
  listaDeNico = [10,50,100]
```

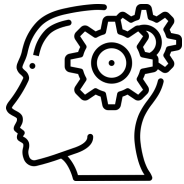
```
✓ [42] doblar_valores(listaDeNico)
```

```
✓ [43] listaDeNico
```

```
[20, 100, 200]
```

Las listas u otras colecciones son del tipo **compuesto**, por lo que se pasa por **referencia**, y las modificamos dentro de la función también lo haremos por fuera.

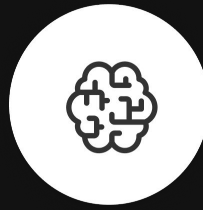
CODER HOUSE



Argumentos Valor - Referencia

Como vimos, las **listas** en este caso, **hacen referencia a su variable original** mientras que los **números** o tipos de datos más simples **“pasan” directamente por valor.**

A continuación una pregunta clave...

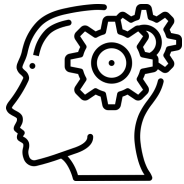


¡PARA PENSAR!

¿Es posible que de alguna forma le digamos a Python cuándo queremos pasar un argumento por referencia o por valor?



RESPONDE EN EL CHAT DE ZOOM



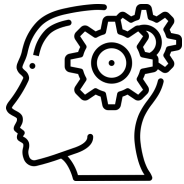
Argumentos Valor - Referencia

La respuesta es **NO**. En Python no se pueden utilizar punteros como en otros lenguajes.

```
>>> def doblar_valor(numero):  
    return numero *= 2  
>>> numero = 10  
>>> numero = doblar_valor(numero)
```

Aunque podemos utilizar trucos, como devolver el valor modificado dentro de la función y volverlo a asignar a la misma variable en caso de desear que sea

“referencia”. 😏



Argumentos Valor - Referencia

En el caso de que sea una colección podemos evitar la modificación directa creando una copia en la llamada. Esto con listas es muy fácil:

```
>>> def doblar_valores(numeros):  
    for i,n in enumerate(numeros):  
        numeros[i] *= 2  
  
>>> numeros = [10, 50, 100]  
  
>>> doblar_valores(numeros[:])
```

Al utilizar **slicing** le indicamos a la función que queremos devolver una copia de la lista desde el principio al fin previniendo la modificación dentro de la función.

ARGUMENTOS INDETERMINADOS



*Uso de *Args y **Kwargs*



¿Para qué se usan?

Lo primero de todo es que en realidad no tienes por qué usar los nombres args o kwargs, ya que se trata de una mera convención entre programadores.



Sin embargo lo que sí debes usar es el asterisco simple `*` o doble `**`.

Es decir, podrías escribir **`*variable` y `**variables`**.



*Uso de *Args*

Gracias a los ***args** en Python, podemos **definir funciones cuyo número de argumentos es variable**. Es decir, podemos definir funciones genéricas que no aceptan un número determinado de parámetros, sino que se “adaptan” al número de argumentos con los que son llamados.



Ejemplo uso de *Args

```
def suma(*args):  
    s = 0  
    for arg in args:  
        s += arg  
    return s
```

```
suma(1, 3, 4, 2)  
#Salida 10
```

```
suma(1, 1)  
#Salida 2
```

Veamos aquí como ***args** puede ser iterado, ya que en realidad es una tupla. Por lo tanto iterando la **tupla** podemos acceder a todos los argumentos de entrada, y en nuestro caso sumarlos y devolverlos.



Ejemplo uso de *Args

```
def suma(*args):  
    return sum(args)
```

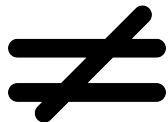
```
suma(5, 5, 3)  
#Salida 13
```

Una forma más sencilla de escribir el
código anterior. 🙌



*Uso de ****Kwargs***

Al igual que en `*args`, en `**kwargs` el nombre es una mera convención entre los usuarios de Python. Puedes usar cualquier otro nombre siempre y cuando respetes el `**`.



A diferencia de `*args`, los `**kwargs` nos **permiten dar un nombre a cada argumento de entrada**, pudiendo acceder a ellos dentro de la función a través de un diccionario.



Ejemplo uso de *Kwargs

```
def suma(**kwargs):  
    s = 0  
    for key, value in kwargs.items():  
        print(key, "=", value)  
        s += value  
    return s
```

```
suma(a=3, b=10, c=3)  
#Salida  
#a = 3  
#b = 10  
#c = 3  
#16
```

Podemos ver que es posible iterar los argumentos de entrada con **items()**, y podemos acceder a la clave **key** (o nombre) y el valor o **value** de cada argumento.



Ejemplo uso de *Kwargs

```
def suma(**kwargs):  
    s = 0  
    for key, value in kwargs.items():  
        print(key, "=", value)  
        s += value  
    return s
```

```
suma(a=3, b=10, c=3)
```

```
#Salida
```

```
#a = 3
```

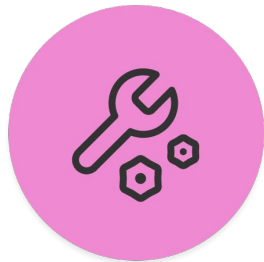
```
#b = 10
```

```
#c = 3
```

```
#16
```

👉 El uso de los `**kwargs` es muy útil si además de querer acceder al valor de las variables dentro de la función, quieres darles un nombre que de una información extra.

Pensar como un diccionario.



RELOJ

Pasaremos de segundos a horas según el parámetro de la función



RELOJ

Tiempo estimado:

Realiza una función que dependiendo de los parámetros que reciba: convierta a segundos o a horas.

- 1- Si recibe un argumento, supone que son segundos y convierte a horas, minutos y segundos.
- 2- Si recibe 3 argumentos, supone que son hora, minutos y segundos y los convierte a segundos.

Funciones Recursivas



Recursividad

La recursión o recursividad es un **proceso de repetición** en el que algo se repite a sí mismo. Es el efecto que sucede cuando se ponen dos espejos frente al otro.

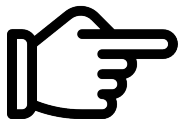
En la informática la recursividad es una **técnica** muy utilizada, la cual se basa en **dividir un problema en partes más pequeñas** para poder solucionarlo de forma más simple.

Donde más se suele utilizar es en las funciones.



Recursividad

Cuando una función se llama a sí misma, tenemos una función recursiva con un comportamiento muy similar al de una sentencia iterativa (if, while, etc) pero debemos encargarnos de planificar el momento en que dejan de llamarse a sí mismas o tendremos una función recursiva infinita.



Podríamos dividir las funciones recursivas en dos:
discursivas sin retorno y discursivas con retorno.

CODER HOUSE



Función recursiva sin retorno



```
def cuenta(numero):  
    numero -=1  
    if numero >0:  
        print(f"---->{numero}")  
        cuenta(numero)  
    else:  
        print("Boooooom!!!")
```

Un ejemplo de una función recursiva sin retorno es el de una **cuenta regresiva** hasta cero a partir de un número dado.

CODER HOUSE



Función recursiva con retorno



```
>>> def factorial(numero):  
    print("Valor inicial ->", numero)  
    if numero > 1:  
        numero = numero *  
factorial(numero - 1)  
    print("Valor final ->", numero)  
    print numero
```

Un ejemplo de una función recursiva con retorno, es el ejemplo del **cálculo del factorial** de un número corresponde al producto de todos los números **desde 1 hasta el propio número**.

CODER HOUSE

Funciones Integradas



Funciones

Ahora que conocemos las funciones, no podemos acabar sin comentar varias de las integradas en Python. Muchas de ellas son para hacer conversiones entre tipos de datos, otras para manipular información, matemáticas, y de más.

👉 Veremos un resumen de las más utilizadas incluyendo algunas ya conocidas.

CODER HOUSE



Int



La función `int()` devuelve un número entero. Es un **constructor**, que crea un entero a partir de un entero float, entero complex o una cadena de caracteres que sean coherentes con un número entero.

```
>>> int(2.5)  
2
```

=

```
>>> int("25")  
25
```

CODER HOUSE



Int



```
>>> int("2.5")
Traceback (most recent call last):
  File "<stdin>", line 1, in
<module>
ValueError: invalid literal for int()
with base 10: '2.5'

>>> int("doscientos")
Traceback (most recent call last):
  File "<stdin>", line 1, in
<module>
ValueError: invalid literal for int()
with base 10: 'doscientos'
```

La función `int()` sólo procesa correctamente cadenas que contengan exclusivamente números.

Si la cadena contiene cualquier otro carácter, la función devuelve una excepción `ValueError`.

CODER HOUSE



Float

```
>>> float(2.5)
2.0
>>> float(25L)
25.0
>>> float("2")
2.0
>>> float("2.5")
2.5
```

La función `float()` devuelve un número coma flotante `float`. Es un **constructor**, que crea un coma flotante a partir de un entero, entero long, entero float (cadenas de caracteres formadas por números y hasta un punto) o una cadena de caracteres que sean coherentes con un número entero.

CODER HOUSE



Str

La función `str()` es el **constructor** del tipo de cadenas de caracteres, se usa para crear un carácter o cadenas de caracteres mediante la misma función `str()`.

Puede convertir un número entero a una cadena de caracteres, de la siguiente forma:

```
>>> str(2.5)
"2.5"
```



Str

👉 Puede convertir un **número float** a una cadena de **caracteres**, de la siguiente forma:

```
>>> str(2.5)
"2.5"
>>> str(-2.5)
"-2.5"
```

👉 Puede convertir un **número complex** a una cadena de **caracteres**, de la siguiente forma:

```
>>> str(2.3+0j)
"(2.3+0j)"
```

👉 Puede convertir un tipo **booleano** a una **cadena de caracteres**, de la siguiente forma:

```
>>> str(True)
"True"
>>> str(False)
"False"
```



Round

La función `round()` redondea un número flotante a una precisión dada en dígitos decimal (por defecto 0 dígitos). Esto **siempre devuelve un número flotante**. La precisión tal vez sea negativa.

👉 En el siguiente ejemplo veremos el redondeo de un número flotante a entero, mayor o igual a .5 al alza:

```
>>> round(2.5)  
3
```

👉 En este otro ejemplo veremos el redondeo de un número flotante a entero, menor de .5 a la baja:

```
>>> round(2.4)  
2
```



Help

Invoca el menú de ayuda del intérprete de Python

```
>>> help()
```

Welcome to Python 3.8! This is the online help utility.

If this is your first time using Python, you should definitely check out the tutorial on the Internet at <https://docs.python.org/3.8/tutorial/>.

Enter the name of any module, keyword, or topic to get help on writing Python programs and using Python modules. To quit this help utility and return to the interpreter, just type "quit".

To get a list of available modules, keywords, or topics, type "modules", "keywords", or "topics". Each module also comes with a one-line summary of what it does; to list the modules whose summaries contain a given word such as "spam", type "modules spam".

```
help>
```

CODER HOUSE



¡Funciones!

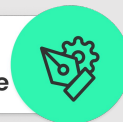
Breve resumen de la consigna del desafío.

¡FUNCIONES!

Formato: Documento de Word, Google Docs o PDF o mejor aún Colabs, con el nombre “Funciones+Apellido”.

Sugerencia: Haz una copia del documento para trabajar.

Desafío
entregable



>> Consigna:

Realizar los ejercicios del siguiente documento disponible en la carpeta:

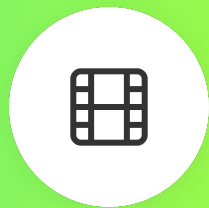
 [Desafío entregable 5 \(Clase 9 y 10\)](#)

>>Aspectos a incluir en el entregable:

Copia del documento con tus respuestas.

¿PREGUNTAS?





***¿QUIERES SABER MÁS? TE DEJAMOS
MATERIAL AMPLIADO DE LA CLASE***



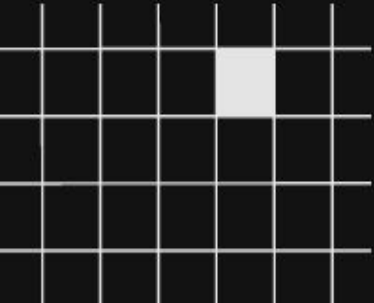
- Artículo: [Funciones](#)
- Artículo: [Funciones Avanzadas](#)
- Artículo: [Funciones Integradas](#)
- Artículo: [Funciones Recursivas](#)
- [RepasoEjercicios](#)





¡MUCHAS GRACIAS!

Resumen de lo visto en clase hoy:

- Argumentos y Parámetros
 - Funciones Recursivas
 - Funciones Integradas
- 



OPINA Y VALORA ESTA CLASE

#DEMOCRATIZANDO LA EDUCACIÓN

CODER HOUSE